Systems programming

14 External data representation

MEEC LEEC MEAer LEAer MEIC-A

João Nuno Silva

TÉCNICO LISBOA

Bibliography

- On HolyWars and a Plea for Peace, Danny Cohen
- The Linux Programming Interface - Michael Kerrisk
  - Section 59.2, 59.3
- Request for Comments 4506 - XDR: External Data Representation Standard
- Mobile Forensics - The File Format Handboo - Christian Hummert
  - Sections 9.1, 9.2
- Protocol Buffers Documentation
  - https://developers.google.com/protocol-buffers/
- Protobuf-c 1.3.3 - Protocol Buffers implementation in C
  - https://github.com/protobuf-c/protobuf-c

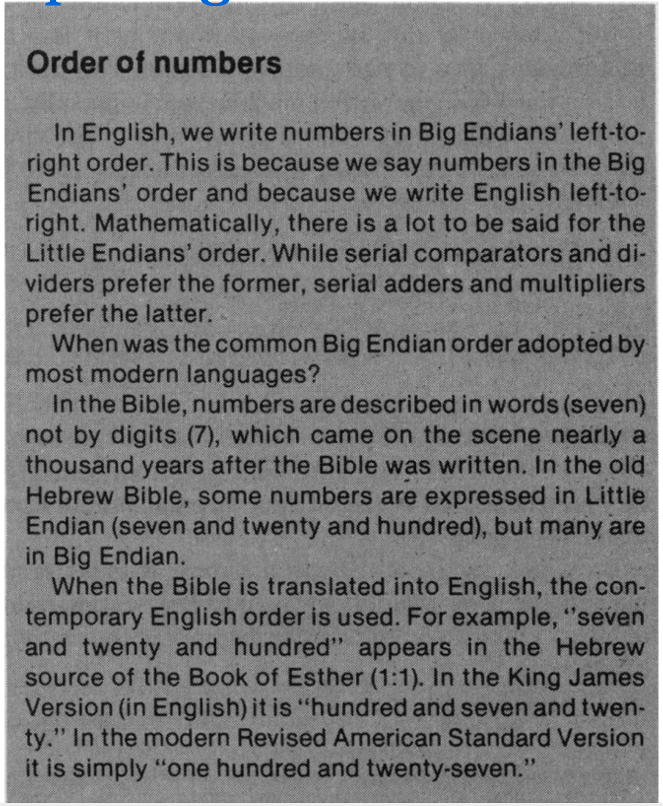
Data Heterogeneity

System data

- Internal data is represented as data structures
  - C structures/arrays
  - Java objects
  - Python objects
- Transferred data is represented as byte sequences
- Data must be flatten to be transmitted
- Same data type can have multiple representations
  - e.g. floats/integers/characters

Endian wars

- https://gwern.net/doc/cs/algorithm/1981-cohen.pdf



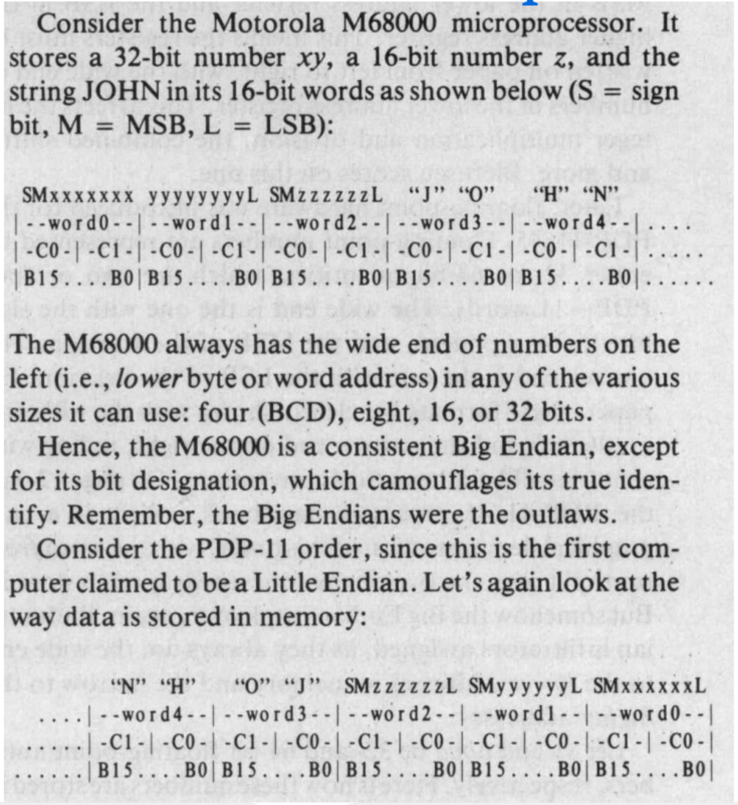
Order of numbers

In English, we write numbers in Big Endians' left-to-right order. This is because we say numbers in the Big Endians' order and because we write English left-to-right. Mathematically, there is a lot to be said for the Little Endians' order. While serial comparators and dividers prefer the former, serial adders and multipliers prefer the latter.

When was the common Big Endian order adopted by most modern languages?

In the Bible, numbers are described in words (seven) not by digits (7), which came on the scene nearly a thousand years after the Bible was written. In the old Hebrew Bible, some numbers are expressed in Little Endian (seven and twenty and hundred), but many are in Big Endian.

When the Bible is translated into English, the contemporary English order is used. For example, "seven and twenty and hundred" appears in the Hebrew source of the Book of Esther (1:1). In the King James Version (in English) it is "hundred and seven and twenty." In the modern Revised American Standard Version it is simply "one hundred and twenty-seven."



Consider the Motorola M68000 microprocessor. It stores a 32-bit number *xy*, a 16-bit number *z*, and the string JOHN in its 16-bit words as shown below (S = sign bit, M = MSB, L = LSB):

```
 SMxxxxxxx yyyyyyyyL SMzzzzzzL  "J" "O"    "H" "N"
|--word0--|--word1--|--word2--|--word3--|--word4--|....
|-C0-|-C1-|-C0-|-C1-|-C0-|-C1-|-C0-|-C1-|-C0-|-C1-|.....
|B15....B0|B15....B0|B15....B0|B15....B0|B15....B0|......
```

The M68000 always has the wide end of numbers on the left (i.e., *lower* byte or word address) in any of the various sizes it can use: four (BCD), eight, 16, or 32 bits.

Hence, the M68000 is a consistent Big Endian, except for its bit designation, which camouflages its true identify. Remember, the Big Endians were the outlaws.

Consider the PDP-11 order, since this is the first computer claimed to be a Little Endian. Let's again look at the way data is stored in memory:

```
       "N" "H"   "O" "J"  SMzzzzzzL SMyyyyyyL SMxxxxxxL
    ....|--word4--|--word3--|--word2--|--word1--|--word0--|
   .....|-C1-|-C0-|-C1-|-C0-|-C1-|-C0-|-C1-|-C0-|-C1-|-C0-|
  ......|B15....B0|B15....B0|B15....B0|B15....B0|B15....B0|
```

Binary byte order

- Byte oder on >16bits numbers depend on the processor architecture
- Can be done in two ways:
  - Big-endian - lower addresses with higher order bits (ex: ARM *).
  - Little-endian - lower addresses with lower order bits (ex: Intel x86).
- Integer 1000465 (0x00 0F 44 11),

Little-endian - 0x000F4411			Big-endian - 0x1144F00	
M[3]	00		M[3]	11
M[2]	0F		M[2]	44
M[1]	44		M[1]	0F
M[0]	11		M[0]	00

Binary byte order

- Not relevant when handling number in I/O
  - Values are typed as text “123”
  - C libraries correctly convert to 4 bytes (big or litle endian)
- Problem if transfered in binary form
  - Memory copy

Little-endian - 0x000F4411			Big-endian - 0x11440F00	
M[3]	00		M[3]	00
M[2]	0F		M[2]	0F
M[1]	44		M[1]	44
M[0]	11		M[0]	11

Integer 1000465 (0x000F4411)

Integer 289672960 (0x44110F00)

Data transmission

- Format of the transmitted data should be agreed
  - conversion to a common format
    - transmitter converts
    - receiver converts
  - transmitted in the sender format
    - receiver converts

Heterogeneity in communication

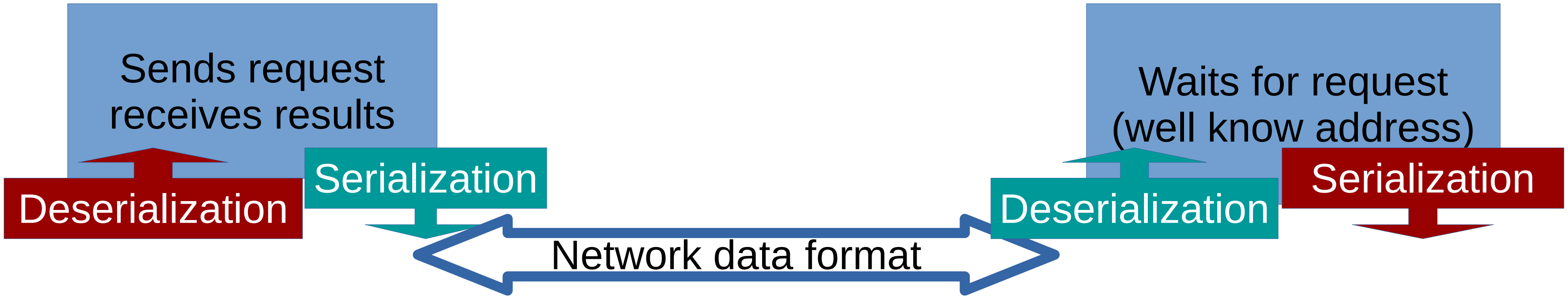
- Integers are just the tip of the portable-data iceberg
- Arbitrary data structures present portability problems
  - alignment
  - Data size
  - pointers.
- Alignment on word boundaries may cause the size of a structure to vary from machine to machine.
- Pointers, which are very convenient to use,
  - have no meaning outside the machine where they are defined.
- Same data structure (int, float) can have different length

How to represent data?

- How to represent data?
- Which data types do you want to support?
  - Base types, Flat types, Complex types
- How to encode data into the wire
- How to decode the data?
  - Self-describing (tags)
  - Implicit description (the ends know)
- Several answers:
  - Many frameworks do these things automatically

Data transfer

- What data to transmit?
- What kind of protocol to use?
- How to encode data?



Data encoding protocols

- Requirements
  - Correction
  - Efficiency
  - Interoperability (language/OS)
  - Ease to use

Network byte order

Network Byte Order Conversion

- Big-endian advantages:
  - Integers are stored in the same order as strings (from left to right).
  - Number signal is on the “first byte” (base address) .
- Little-endian advantages:
  - Eases conversion between different length integers (ex: 12 is represented by 0x0C or 0x000C).
- In the Internet,
  - Addresses are always big-endian.
  - The first ARPANET routers/Interface Message Processor were 16 bits Honeywell DDP-516 computers
    - Implemented big-endian

Network Byte Order Conversion

- Current internet has devices using
  - Big-endian and little endian
- Routers and switches and network packets
  - big-endian
- Intel 386, pentium, … family
  - Little-endian
- ARM
  - Big-endian
- RISC-V
  - Big-endian

Network Byte Order Conversion

- Conversion of 16 bit and 32 bit integers is necessary when
  - When sending data to the network
  - Receiving data from the network

host byte order to network byte order.

- unsigned integer long
  - uint32_t htonl(uint32_t hostlong);
- unsigned short integer
  - uint16_t htons(uint16_t hostshort);
-

network byte order to host byte order

- unsigned integer long
  - uint32_t ntohl(uint32_t netlong);
- unsigned short integer
  - uint16_t ntohs(uint16_t netshort);

Network Byte Order Conversion

- Limitations
  - Only aconverts two basic types:
    - 16 and 32 bit integers
  - Requires programmer to explicitly converter every value
  - Sender and recipient call different functions
    - htns → nths
- Impossible to easily convert complex data
  - Structures, arrays, ...
- Data in different languages difficult to convert

External Data Representation

External Data Representation

- XDR tries to standardizing data representations
  - Defines a canonical data representation
  - XDR defines a single byte order (Big Endian)
  - XDR defines a single floating-point representation (IEEE)
  - XDR defines how to serialize complex data
    - Strcutures, arrays, dynamic lists, ...

XDR

- The single standard decouples programs
  - that create or send portable data from those that use or receive portable data
- New machine or a new language has no effect on existing portable programs.
  - A new machine joins by being “taught” how to convert between standard representations and its local representations;
- With XDR programmers only call a set of functions
  - To serialize any data type

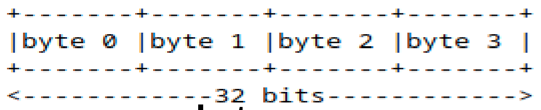
XDR

- The local representations of other machines are irrelevant to existing programs running on other machines
  - Machines with different endianess, and data representation
  - Transparent interaction
- Local representations of the other machine is irrelevant
  - Other programs use XDR to read portable data produced
- All data that is transmitted conforms
  - to the canonical standards defined by XDR

XDR canonical datatypes

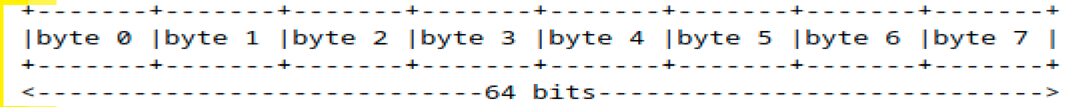
- Basic data
  - Integer / Unsigned Integer
  - Hyper Integer and Unsigned Hyper Integer

ing-



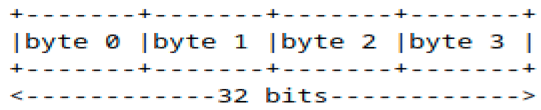
```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

Integer



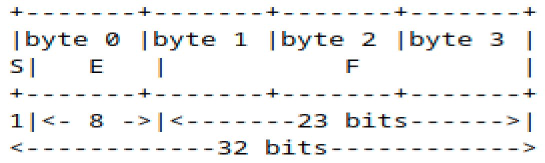
```
+-------+-------+-------+-------+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |byte 4 |byte 5 |byte 6 |byte 7 |
+-------+-------+-------+-------+-------+-------+-------+-------+
<----------------------------64 bits---------------------------->
```

Hyper (Unsigned) Integer



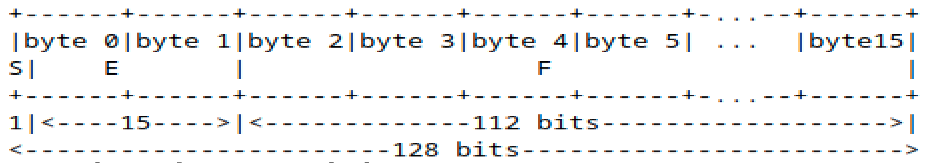
```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

Unsigned Integer



```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
S|   E   |           F          |
+-------+-------+-------+-------+
1|<- 8 ->|<-------23 bits------>|
<------------32 bits------------>
```

Single Precision FP

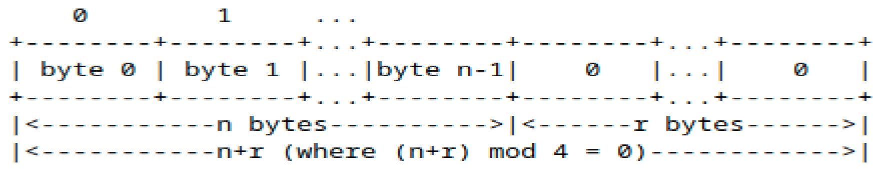


```
+------+------+------+------+------+------+-...--+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5| ...  |byte15|
S|    E       |                  F                      |
+------+------+------+------+------+------+-...--+------+
1|<----15---->|<-------------112 bits------------------>|
<-----------------------128 bits------------------------>
```

Qudruple Precision FP

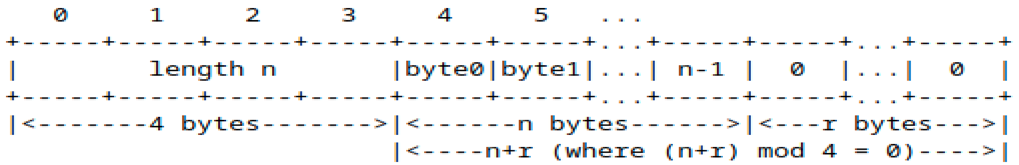
XDR canonical datatypes

- Variable size data data
  - Fixed-Length Opaque Data
  - Variable-Length Opaque Data
  - String



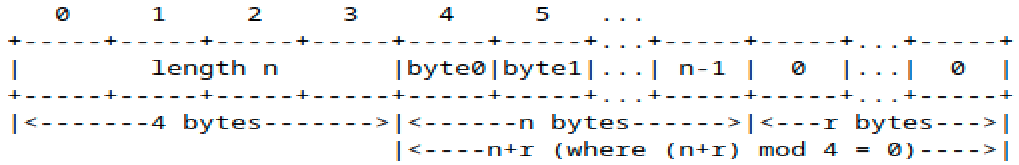
```
        0        1     ...
    +--------+--------+...+--------+--------+...+--------+
    | byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
    +--------+--------+...+--------+--------+...+--------+
    |<-----------n bytes---------->|<------r bytes------>|
    |<-----------n+r (where (n+r) mod 4 = 0)------------>|
```

Fixed length opaque data



```
      0     1     2     3     4     5   ...
    +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
    |        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
    +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
    |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                            |<----n+r (where (n+r) mod 4 = 0)---->|
```

variable length opaque data

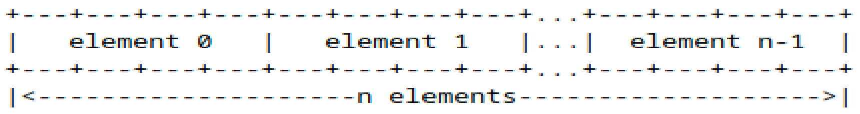


```
      0     1     2     3     4     5   ...
    +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
    |        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
    +-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
    |<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
                            |<----n+r (where (n+r) mod 4 = 0)---->|
```

String

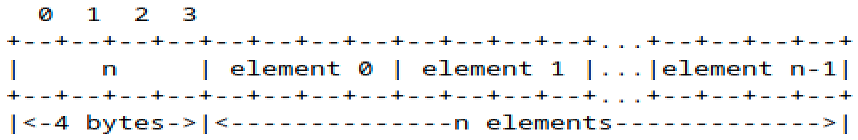
XDR canonical datatypes

- Composed data
  - Fixed-Length Array
  - Variable-Length Array
  - ted U



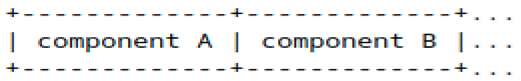
```
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|   element 0   |   element 1   |...|  element n-1  |
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|<--------------------n elements------------------->|
```

Fixed length Array



```
  0   1   2   3
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|     n     | element 0 | element 1 |...|element n-1|
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|<-4 bytes->|<--------------n elements------------->|
```

Variable length array



```
+-------------+-------------+...
| component A | component B |...
+-------------+-------------+...
```

Structure

XDR Library

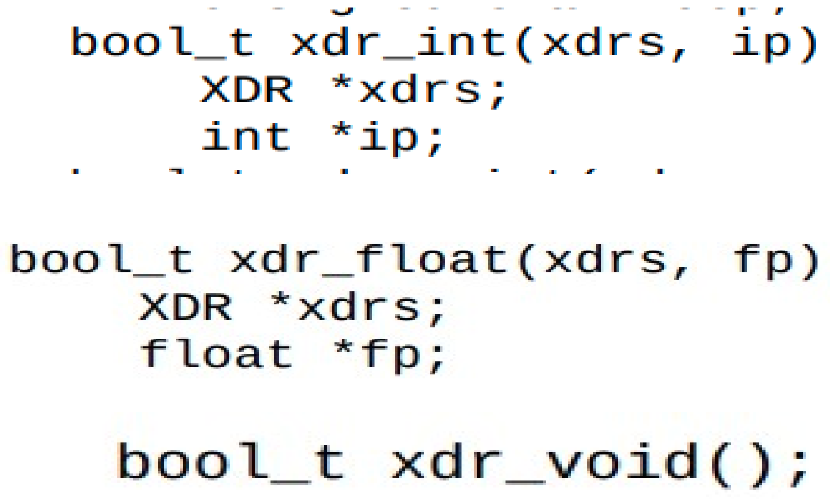
- XDR library not only solves data portability problems
  - Transformation to/from canonical format
- Allows to write/read arbitrary C constructs
  - consistent, specified, well-documented manner.
- Composed of set of functions
  - To encode/decode data (same function)
  - Based on defined data structures
  - Called filters

XDR filters

- Function that receive a stream and converts data
  - Same function is used to serialize and deserialize data
- Stream can be of output (write) or input (read)
  - If stream is of output
    - Data is converted from host to canonical
    - And put on the stream
  - If stream is of input
    - Data is read from the stream
    - And converted from canonical to host
- Data is received as an argument and error code is returned

XDR filters - Basic data filters

- Number Filters
  - Char, short, int, long,
- Floating Point Filters
  - Float, double, ...
- No data
  - void



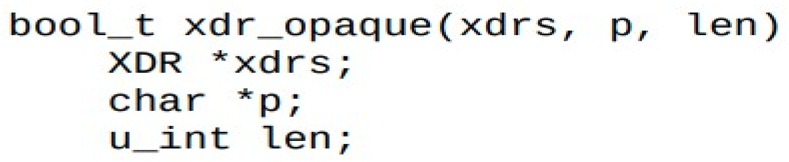
```
bool_t xdr_int(xdrs, ip)
      XDR *xdrs;
      int *ip;
```

```
bool_t xdr_float(xdrs, fp)
      XDR *xdrs;
      float *fp;
```

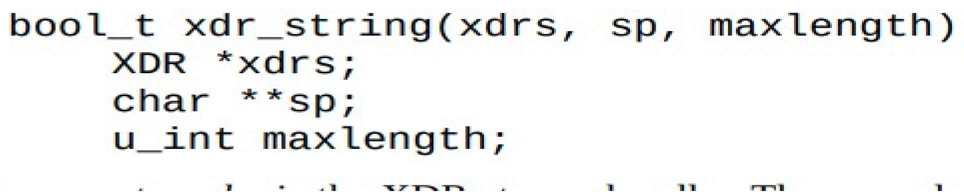
```
bool_t xdr_void();
```

XDR filters- Variable size data

- Fixed-Length Opaque Data
- Variable-Length Opaque
- String



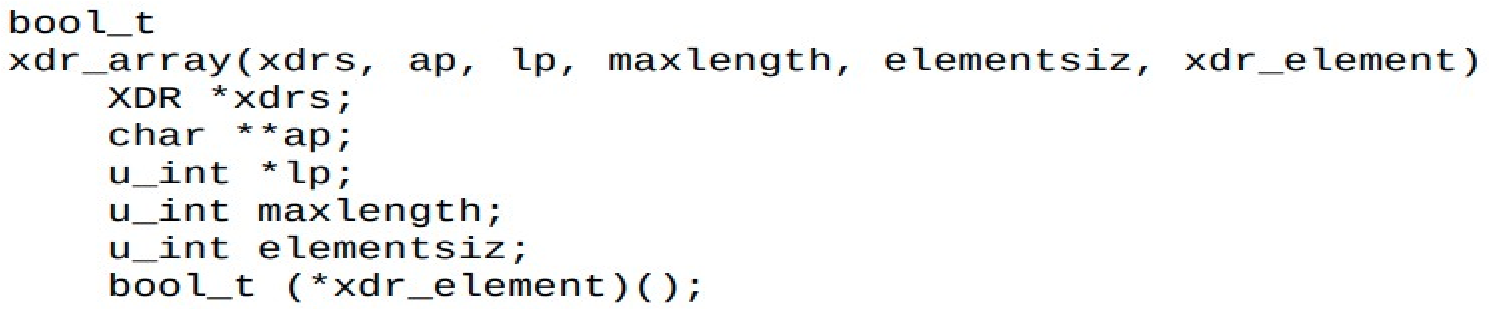
```
bool_t xdr_opaque(xdrs, p, len)
    XDR *xdrs;
    char *p;
    u_int len;
```



```
bool_t xdr_string(xdrs, sp, maxlength)
    XDR *xdrs;
    char **sp;
    u_int maxlength;
```

XDR filters - Composed data

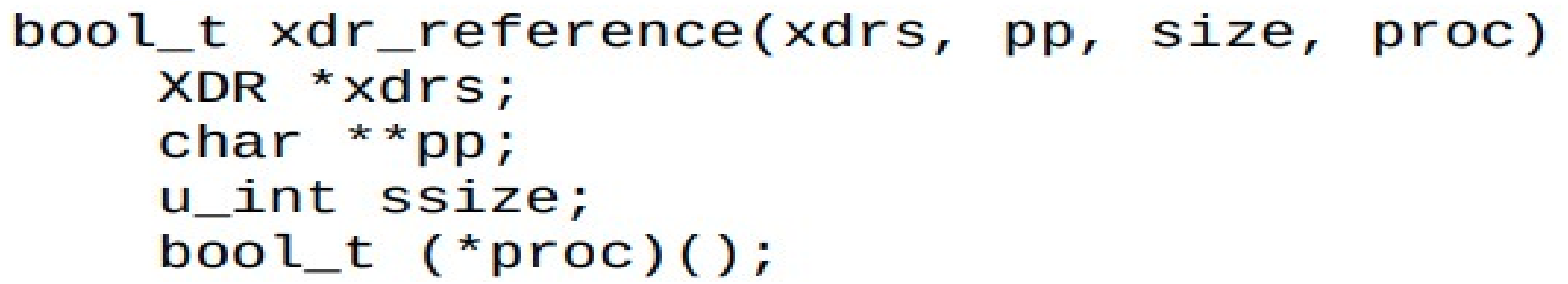
- Fixed-Length Array
- Variable-Length Array
- Struct



```
bool_t
xdr_array(xdrs, ap, lp, maxlength, elementsiz, xdr_element)
    XDR *xdrs;
    char **ap;
    u_int *lp;
    u_int maxlength;
    u_int elementsiz;
    bool_t (*xdr_element)();
```

XDR filters - pointers

- Filter knows how to transform * pp
  - Proc converts pointed data
- When reading



```
bool_t xdr_reference(xdrs, pp, size, proc)
     XDR *xdrs;
     char **pp;
     u_int ssize;
     bool_t (*proc)();
```

How to use XDR in data transfer

- XDR data strutures do not contain meta-data
  - Impossible to know type from binary data
- Client and server must
  - Agree on the format of transferred data
    - XDR files from RPC
  - Have compatible encoder/decoder
    - Same code encodes/decodes data
  - Call encoding/decoding function in same order

How to use XDR in data transfer

```
  struct gnumbers {
      long g_assets;
      long g_liabilities;
  };
xdr_gnumbers(XDR *xdrs, struct gnumbers *gp){
   return(xdr_long(xdrs, &gp->g_assets) &&
       xdr_long(xdrs, &gp->g_liabilities));
}
```

Protocol Buffers

Google Protocol Buffers

- Goals
  - provide a language and platform-neutral way to specify and serialize data
  - Serialization should be efficient
  - Serialization should simple to use
  - Serialized data can be stored or transmitted over the network
- Protocol Buffers provides a language to specify messages
  - To allow automatically generated serialization code

Serialization code generation

- Serialization code is
  - Repetitive
  - To well know data types follows well know patterns
- Many systems generate serialization code from messages specification
- IDL – Interface Description Language
  - Describes an interface/exchanges messages
  - Separates logical description of data from
    - Programming language
    - Serialization code
    - Data wire format

Google Protocol Buffers

- Efficient, binary serialization
- Support protocol evolution
  - Can add new messages
  - Allow changes in messages
- Supports types
  - That generate compilation errors when defining the messages data
- Supports complex structures
-

Google Protocol Buffers

- Programmers define a new “message” types
  - In the .proto file
  - Definition similar to C structures
    - Extra parameters
- Special compiler processes such .proto file
  - Generates all the functions to serialize/deserialize the messages
  - Generates a library for programmer to use
- Different programming languages have different compilers
  - Generate different libraries, but all compatible with each other
- Google uses them everywhere (50k proto buf definitions)

Protocol Buffer Language

- Message contains uniquely numbered fields
- Field is represented by
  - field-type,
  - Data-type
  - Field-name
  - encoding-value
  - default value
- Filed-Name
  - Used by the programmer to access the data
- Encoding-value
  - Use in the serialized message to identify the specific field

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

Protocol Buffer Language

- Available data-types
  - Primitive data-type
  - int, float, bool, string, raw-bytes
  - Enumerated data-type
  - Nested Message
    - Allows structuring data into an hierarchy
- Field-types can be:
  - Required fields
  - Optional fields
  - Repeated fields
  - Dynamically sized array

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
  repeated PhoneNumber phone = 4;
}
```

.proto file

- The specification of the message is contained in a .proto file
- The .proto file is compiled by protoc tool
  - protoc-c --c_out=. amessage.proto
  - The output of the protoc is a generated code that allows programmers to manipulate the particular message type
- .h and .c files that contain all the necessary finctions to:
    - Put data in the message structure
    - Serialize the message structure
    - De serialize the message structure
    - get data from the message structure

.proto file

- Java
- Class **Person** with
  - Hidden fields
  - Public methods

        boolean hasId();
        int getId();
        Builder setId(int value)

    - ...
- import protos.Person;

- C
- Structure **person**
  - With fields

    Person__PhoneNumber **phone;
  - ...
- Functions
  - person__get_packed_size()
  - message__pack()
  - message__unpack();
- #include "../example.pb-c.h"

Data types

- Messages
  - structures/classes
- Numbers
  - Float (long / double)
  - Integer (32 / 64bits)
- Booleans
- Enumerates
- Lists
- Maps
- Nested messages

- Mapped to all languages
  - From C to Python
- Values accessed as regular data
- Efficiently serialized and transmitted

Protocol Buffer Language

- Produced libraries facilitate programming
  - Uses languages facilities
    - Datatype, classes
  - Verifies data
    - Type of values
    - Whether were set
  - Facillitates messages memory management
    - Programmer knows exactly number of bytes to transmit